Towards Validating a Platoon of Cristal Vehicles using CSP||B*

Samuel Colin[1], Arnaud Lanoix[1], Olga Kouchnarenko[2], and Jeanine Souquières[1]

[1] LORIA – DEDALE Team – Campus scientifique
F-54506 Vandoeuvre-Lès-Nancy, France
{firstname.lastname}@loria.fr
[2] LIFC – TFC Team – 16 route de Gray
F-25030 Besançon, France
{firstname.lastname}@lifc.univ-fcomte.fr

Abstract. The complexity of specification development and verification of large systems has to be mastered. In this paper a specification of a real case study, a platoon of Cristal vehicles is developed using the combination, named CSP||B, of two well-known formal methods. This large – both distributed and embedded – system typically corresponds to a multi-level composition of components that have to cooperate. We show how to develop and verify the specification and check some properties in a compositional way. We make use of previous theoretical results on CSP||B to validate this complex multi-agent system.

1 Introduction

This paper is dedicated to the validation of land transportation systems taken as an application domain. These systems, which are both distributed and embedded, require to express functional as well as non functional-properties, for example time constrained response and availability of required services. As with any distributed system, a component assembly may appear obscure behaviours that are hard to understand and difficult to debug. As with any embedded system, components and their composition should satisfy safety/security/confidence requirements. As component-based systems are omnipresent, it is important to ensure their correct assembly.

Our goal is to apply the CSP||B combination [1] of well-established formal methods, CSP [2] and B [3], to a specific distributed and embedded system. This case study is a convoy of so-called Cristal vehicles seen as a multi-agents system. We motivate the use of this CSP||B combination by the existence of pure B models describing the agents and vehicles behaviours [4]. By using CSP for composing B machines we aim at giving these B models the architectural, compositional description they lack.

As a comparison point, in [1] Schneider & Treharne illustrate their use of CSP||B with a multi-lift system that can be seen as a distributed system using several instances of a lift, minus the fact that the interactions of the lifts are actually centralised in a

* This work has been partially supported by the French National Research Agency (ANR)/ANR-06-SETI-017 TACOS project, (http://tacos.loria.fr), and by the pôle de compétitivité Alsace/Franche-Comté/CRISTAL project (http://www.projet-cristal.org).

dedicated dispatcher. Our goal is very similar, but in contrast to [1], we want to avoid relying on a centralised, or orchestrating, controller.

Similar works exist on structured development with the B method using decomposition, hence in a more "top-down" approach, and refinement. For instance, Bontron & Potet [5] propose a methodology for extracting components out of the enrichments brought by refinement. The extracted components can then be handled to reason about them so as to validate new properties or to detail them more. The interesting point is that their approach stays within the B method framework: this means that the modelling of component communication and its properties has to be done by using the B notation, which can quickly get more cumbersome than an ad-hoc formalism like CSP. Abrial [6] introduces the notion of decomposition of an event system: components are obtained by splitting the specification in the chain of refinements into several specifications expressing different views or concerns about the model. Attiogbé [7] presents an approach dual to the one of Abrial: event systems can be composed with a new asynchronous parallel composition operator, which corresponds to bringing "bottom-up" construction to event systems. In [8], Bellegarde & al. [8] propose a "bottom-up" approach based on synchronisation conditions expressed on the guards of the events. The spirit of the resulting formalism is close to that of CSP‖B. Unfortunately it does not seem to support message passing for communication modelling.

Our approach is rather "bottom-up" oriented: the B machines describe the various components of a Cristal vehicle while CSP is used for expressing their assembly at the level of a single vehicle and at the level of the whole convoy[3]. Our experience, reported here, shows that writing and checking CSP‖B specifications can help eliminate errors and ambiguities in an assembly and its communication protocols. We also believe that writing formal specifications can aid in the process of designing autonomous vehicles.

This paper is organised as follows. Section 2 introduces the platooning case study with the properties we will focus on. Section 3 briefly introduces the theoretical background on CSP‖B. Section 4 presents the specification and the verification process of a single Cristal vehicle whereas Section 5 is dedicated to a platoon of vehicles. Section 6 ends with some perspectives of this development.

2 Case Study Presentation : a Platoon of Vehicles

The CRISTAL project aims at the development of a new type of urban vehicle with new functionalities and services. One of its major cornerstones is the development, certification and validation of a platoon of vehicles.

A platoon is a set of autonomous vehicles which have to move in a convoy, i.e. following the path of the leader (possibly driven by a human being) in a row. Its control concerns both a longitudinal control, i.e. maintaining an *ideal* distance between each vehicle, and a lateral control, i.e. each vehicle should follow the track of its predecessor. Both controls can be studied independently [9]. In the sequel, we will only focus on the longitudinal one.

[3] CSP‖B specifications discussed in this paper are available at `http://www.loria.fr/~lanoix/platoon.zip`.

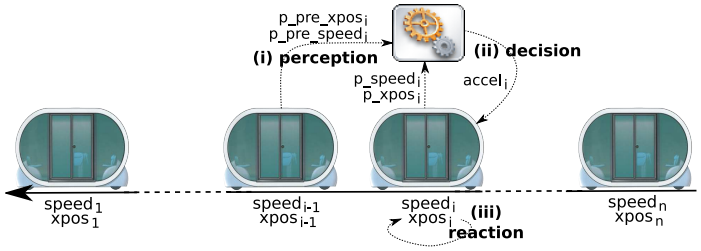

Fig. 1. A platoon of Cristals

Through the projects' collaborations, we have decided to consider each vehicle, named Cristal in the following, as an agent of a Multi-Agent System (MAS). The Cristal driving system perceives information about its environment before producing an instantaneous acceleration passed to its engine. In this context, we consider the platooning problem as a situated MAS which evolves following the Influence/Reaction model (I/R) [10] in which agents are described separately from the environment.

As we only focus on the longitudinal control of the platoon, the considered space is one-dimensional. Hence the position of the i^{th} Cristal is represented by a single variable $xpos_i$, its velocity by $speed_i$. The behaviour of the Cristal controllers can be summarised as follows, see Fig. 1:

- **(i) perception step:** each Cristal driving system receives its velocity p_speed_i and its position p_xpos_i, from the physical part of the Cristal. Furthermore, it receives by network communication the velocity $p\_pre\_speed_i$ and the position $p\_pre\_xpos_i$ of its leading Cristal
- **(ii) decision step:** each Cristal driving system can influence its speed and position by computing and sending to its engine an instantaneous acceleration $accel_i$. The acceleration can be negative, corresponding to the braking of the Cristal
- **(iii) reaction step:** $xpos_i$ and $speed_i$ are updated, depending on the current speed $speed_i$ of the Cristal and a decided instantaneous acceleration $accel_i$ of the engine

Our goal is the expression of the model with a broader range of granularity than the existing B model [4]. Our CSP||B model should span more architectural levels (from the component of a vehicle to the whole convoy) and explicitly model communications. It is thus necessary to ensure that communications between components in the resulting architecture do not suffer from design errors, e.g. a scheduling leading to deadlocks.

3 Theoretical Background on CSP||B

The B machines specifying components are open modules which interact by the authorised operation invocations. When developing distributed and concurrent systems, CSP is used to describe an execution order for invoking the B machines operations. CSP describes processes – objects or entities which exist independently, but may communicate

with each other. There is a lot of works on CSP‖B. The reader interested by theoretical results is referred to [1,11,12]; for case studies, see for example [13,14].

3.1 CSP Controllers

In the combined CSP‖B model, the B part is specified as a standard B machine[4] without any restriction on the language, while a controller for a B machine is a particular kind of CSP process, called a CSP controller.

CSP controllers obey the following (subset of the) CSP grammar:

$$\begin{array}{l} P ::= c\ ?\ x\ !\ v\ \rightarrow P \mid ope\ !\ v\ ?\ x \rightarrow P \mid \\ \qquad b\ \&\ P \mid P1 \,\Box\, P2 \mid S(p) \end{array}$$

The process $c\ ?\ x\ !\ v \rightarrow P$ can accept input x and output v along a communication channel c. Having accepted x, it behaves as P. To interact with a B machine, a controller makes use of *machine channels* which provide the means for controllers to synchronise with the B machine. For each operation $x \longleftarrow ope(v)$ of a controlled machine, there is a channel $ope\ !\ v\ ?\ x$ in the controller corresponding to the operation call: the output value v from the CSP description corresponds to input parameter of the B operation, and the input value x corresponds to the output of the operation. A controlled B machine can only communicate on the machine channels of its controller.

The behaviour of a guarded process $b\ \&\ P$ depends on the evaluation of the boolean condition b: if true, it behaves as P, otherwise it is unable to perform any events. In some works (e.g. [1]), the notion of *blocking assertion* is defined by using a guarded process on the inputs of a channel to restrict these inputs: $c\ ?\ x\ \&\ E(x) \rightarrow P$. The external choice $P1 \,\Box\, P2$ is initially prepared to behave either as $P1$ or as $P2$, with the choice made on the occurrence of the first event. The expression $S(p)$ is a recursive process invocation.

In addition to the language for simple processes, CSP provides a number of operators to combine processes. In this paper the operators we are concerned with are $P1 \parallel_E P2$, and $|||_i(P(i))$.

- The sharing operator $P1 \parallel_E P2$ executes $P1$ and $P2$ concurrently, requiring that $P1$ and $P2$ synchronise on the events into the sharing alphabet E and allowing independent executions for other events (not in E)[5].
- The indexed form of the interleaving operator $|||_i P(i)$ executes the processes $P(i)$ in an independent manner without synchronisation. It is used to build up a collection of similar processes independent from each other.

[4] Because of lack of space, we only recall the idea behind consistency checking in the B method. Roughly speaking, given a B machine and its invariant, the machine is said *consistent* if its initialisation satisfies the invariant, and if, for each operation, assuming its precondition and the invariant hold, the operation body satisfies the invariant.

[5] Note that when combining a CSP controller P and a B machine M associated with$P$, the sharing alphabet can be dropped: $(P \parallel_{\alpha(M)} M) \equiv P \parallel M$.

As for other process algebras, the denotational semantics of CSP is based on the observation of process behaviours. In CSP, the three main semantic models use notions of traces, stable failures, and failures/divergences (see [15]). In the trace semantics, a process P is associated with the set of finite sequences of events that P can perform, denoted traces(P). In the stable failures semantics, a process P is associated with the set failure (P) of pairs of the form (tr , X), where tr is a finite trace in traces(P), and X is the set of events that P cannot perform after the execution of the events of tr. This model allows specifying the deadlocks of P. Finally, in the failures/divergences semantics, a process P is associated with the set of its stable failures, and with the set of its divergences. The process P is said divergent if it is in a divergent state where the only possible events are internal (or invisible) events. The divergences set of P, denoted divergences(P), is the set of traces tr such that P is in a divergent state after performing events of tr.

The three most frequently used CSP refinement notions compute and compare the semantic models of processes. Given two processes P and Q, we say

- P $\sqsubseteq_T$ Q, Q refines P in the trace semantics if all the possible communication sequences that Q may perform, are also possible sequences for P;
- P $\sqsubseteq_F$ Q, Q refines P in the stable failure semantics if failure (Q) $\subseteq$ failures (P);
- P $\sqsubseteq_{FD}$ Q, Q refines P in the failures/divergences model if failures (Q) $\subseteq$ failures (P) and divergences(Q) $\subseteq$ divergences(P).

The FDR2 model checker [16] provides determining deadlock and divergence freedom of individual CSP processes, and implements verification for each kind of refinement.

3.2 Useful Results on CSP||B

The main problem with combined specifications is the model consistency, in other words, CSP and B parts should not be contradictory. To ensure the consistency of a controlled machine (P||M) in CSP||B, a verification technique has been proposed [12] consisting in verifying the following sufficient conditions:

- the divergence-freedom of (P||M);
- the deadlock-freedom of P.

This verification technique can be generalised to a set of controlled machines (Pi || Mi) evolving in parallel:

- the divergence-freedom of each (Pi || Mi);
- the deadlock-freedom of (P1||P2|| ... || Pn).

The divergence-freedom of (P||M) can be deduced by using a technique based on *Control Loop Invariants* (CLI). This technique involves the verification that each path a controlling process may take, does not end up in a diverging state (a violation of the precondition of a controlled method, for instance). For verifying that we reuse the methodology introduced in [11]. It involves the translation of the various paths of the controlling process up to recursive calls to itself, into B operations in a machine. This

machine is then augmented with a CLI, and this machine consistency checking is performed: it is thus akin to verify that no path in the controlling process ends up in a diverging state.

Let S(p) be a family of processes in a controller P, p helping to identify which process we are referencing to. S(p) is of the following general form:

$S(p) = path\_1 \rightarrow S(q) \Box \ ... \ \Box \ path\_n \rightarrow S(r)$

Let $BBODY_{S(p)}$ be the rewriting of S(p) into B using the translation rules of [11]. The whole controlling process P is then translated into a B machine, whose methods are the various $BBODY_{S(p)}$ and whose invariant constitutes the chosen CLI.

If the rewriting of P into a B machine is consistent, it means all the operations preserve the invariant. This in turn means that each process of the controller forms a sequence of operation calls that maintain the CLI. This entails that the controller never diverges in calling its controlled B machine, hence that the couple controller/B machine is divergence-free. This is the matter of the following theorem:

Theorem 1 ([12, Theorem 1]). *If CLI is a predicate such that*

$$CLI \wedge I \Rightarrow [BBODY_{S(p)}]\ CLI$$

for each $BBODY_{S(p)}$ in P, then (P‖M) is divergence-free.

The following result is useful for establishing trace properties of controlled components. It means that the trace refinement established purely for the CSP part (possibly using hidden events) of a controlled component suffices to ensure the trace refinement for the overall controlled component (possibly using hidden events).

Corollary 1 ([1, Corollary 7.2]). *For any controller P and any B machine M with the alphabet α(M) of events one has:*
1. If $S \sqsubseteq_T P$ then $S \sqsubseteq_T (P \| M)$
2. If $S \sqsubseteq_T P \setminus E$ and $E \subseteq \alpha(M)$, then $S \sqsubseteq_T (P \| M) \setminus E$

The following theorem is a composition result for establishing the whole system divergence-freedom from the divergence-freedom of its components.

Theorem 2 ([1, Theorem 8.1]). *If (Pi ‖ Mi) is divergence-free for each i, then $\|_i (Pi \| Mi)$ is divergence-free.*

The consistency of a single controlled machine is achieved by the following result stating that the deadlock-freedom of (P‖M) can be deduced by establishing the deadlock-freedom of the P part.

Theorem 3 ([1, Theorem 5.9]). *If P is a CSP controller for M with no blocking assertion on any machine channels of M, and P is deadlock-free in the stable failures model, then (P‖M) is deadlock-free in the stable failures model.*

Finally, the deadlock-freedom of multiple controlled machines $\|_i (Pi \| Mi)$ follows from deadlock-freedom of the combination of the CSP parts $\|_i Pi$. It achieves the multiple controlled machines consistency checking.

Theorem 4 ([1, Theorem 8.6]). *Given a collection of CSP controllers Pi and corresponding B machines Mi, such that no controller has any blocking assertions on the control channels: then if $\|_i Pi$ is deadlock-free in the stable failures model, then so too is $\|_i (Pi \| Mi)$.*

4 Specifying a Single Cristal

We consider a Cristal vehicle composed of two parts: its engine and a driving system, as depicted Fig. 2. Each part is itself built upon a B machine controlled by an associated CSP process.

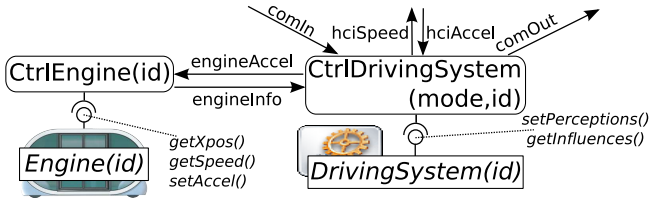

Fig. 2. Architectural view of a Cristal

We must ensure steady communications between Cristal components. For instance, communications are broken if two components expect input from each other: in that case the components cause the deadlock of the whole vehicle. We therefore state that the communications between Cristal components should never cause a deadlock.

In an automatic mode, a Cristal must get information about its position and its speed as accurate as possible so that its resulting acceleration is as accurate as possible. Thus we do not want the Cristal to stay in the "perception mode" for too long. To avoid that, a solution is to force the Cristal to alternate between "perception mode" and "reaction mode". This is what we strive for as a safety property.

The engine is built upon a B machine that describes its inner workings, i.e. its knowledge of speed and position as well as how it updates them w.r.t. a given acceleration. The speed and the position are passed on to the controller through the getSpeed and getXpos methods/events. The acceleration is passed on to the engine through the setAccel method/event. The CtrlEngine CSP controller receives acceleration orders through the channel associated with the engineAccel event and sends information about speed and position through the engineInfo event.

In a similar way, the driving system is composed of the DrivingSystem B machine and its CtrlDrivingSystem CSP controller. The machine and its controller share the setPerceptions and getInfluences events. The controller (and thus the compound construction) communicates with the engine through the engineInfo,engineAccel channels. It also communicates with the Human Control Interface (HCI) of the Cristal by way of the hciSpeed, hciAccel events. Finally, it is also able to receive information from or to send information to other Cristals through the comIn and comOut events, respectively.

The models of the engine and the driving system assume a common set of constants. The constants below are replicated in both CSP and B specifications:

- The functioning modes of the Cristals: as a leader (LEADER), as a single vehicle (SOLO) or in a platoon (PLATOON);
- Maximal and minimal allowed accelerations (MAX_ACCEL,MIN_ACCEL);

– Maximal speed (MAX_SPEED), the minimal allowed speed being 0;
– A set of unique identifiers for the Cristals (Ids).

We will now detail each component and the performed verifications.

4.1 The Engine

```
MODEL Engine(Id)
VARIABLES
  speed, xpos
OPERATIONS
  speed0 ⟵ getSpeed = /*...*/
  xpos0 ⟵ getXpos = /*...*/
  setAccel(accel) =
    PRE
      accel ∈ MIN_ACCEL..MAX_ACCEL
    THEN
      ANY new_speed
      WHERE new_speed = speed + accel
      THEN
        IF (new_speed > MAX_SPEED)
        THEN
          xpos := xpos + MAX_SPEED
          || speed := MAX_SPEED
        ELSE
          IF (new_speed < 0)
          THEN
            xpos := xpos − (speed × speed) / (2 × accel)
            || speed := 0
          ELSE
            xpos := xpos + speed + accel / 2
            || speed := new_speed
          END
        END
      END
    END
```

Fig. 3. The Engine(Id) B model

As stated earlier, the engine is a behavioural component reacting to a given acceleration for speeding up or slowing down a Cristal vehicle. This behaviour is described by a Engine(Id) B machine illustrated in Fig. 3. Id[6] is a natural number that uniquely identifies a Cristal. It is used at the CSP level in order to model interactions with other Cristals.

The speed ⟵ getSpeed() and xpos ⟵ getXpos() methods capture data from the engine to pass them on to whomever needs it (say, the HCI, for instance). The setAccel(accel) method models how the Cristal behaves when passed a new instantaneous acceleration.

The B machine is made able to communicate by adding a CSP model for controlling it. This model, called CtrlEngine(id) and depicted in Fig. 4, schedules the calls to its various methods. The getSpeed ? speed and getXpos ? xpos event calls the homonymous methods of the B machine to retrieve the speed and the position of the Cristal. Similarly, the controller passes a new instantaneous acceleration on through setAccel ! accel to the B machine.

```
CtrlEngine_perceptions(id) =
  getXpos ? xpos → getSpeed ? speed → engineInfo.id ! xpos ! speed → CtrlEngine_actions(id)
  □
  getSpeed ? speed → getXpos ? xpos → engineInfo.id ! xpos ! speed → CtrlEngine_actions(id)

CtrlEngine_actions(id) =
  engineAccel.id ? accel → setAccel ! accel → CtrlEngine_perceptions(id)

CtrlEngine(id) = CtrlEngine_perceptions(id)
```

Fig. 4. The CtrlEngine(id) CSP controller

[6] In the whole model we use *id* but as it is a reserved keyword in B we have to resort to denoting it *Id* for the B machines.

Communications are achieved with the engineInfo and engineAccel events. The former sends the current speed and position to requesting components. The latter sets a new acceleration for the engine. The event engineInfo.id ! xpos ! speed has a Cristal identifier as a synchronisation channel and the Cristal position and speed as output channels. Similarly, the event engineAccel.id ? accel has also a Cristal identifier as a synchronisation channel and an acceleration as an input channel.

The protocol defined by the controller is very simple: either it asks the machine about the speed and the position (in any order) and passes it on the engineInfo event, or sets a new acceleration passed on by the engineAccel event. Information request and acceleration setting alternate: CtrlEngine_perceptions calls CtrlEngine_actions which in turn calls CtrlEngine_perceptions again.

The whole engine component is then defined as the composition of the Engine(id) machine and its CtrlEngine(id) controller for a given Cristal identifier id:

$$(\mathsf{CtrlEngine(id)} \parallel \mathsf{Engine(id)})$$

Verification. The Engine(Id) B machine consistency is successfully checked using the B4Free proof tool. The CtrlEngine(id) controller *deadlock-freedom* (in the stable failures model) and its *divergence-freedom* are successfully checked. These verifications have been done with the FDR2 model-checking tool

The composition of the B machine and the controller is verified for *divergence-freedom*. The verification is specific to CSP||B and is not supported by tools and is described in Theorem 1. As the verification involves the translation of the CSP process to B, we illustrate the translation of CtrlEngine(id) in Fig.5. Its CLI is actually as simple as the $\top$ predicate modulo the mandatory typing predicates.

Once all these properties are established, we can use the theorems of Sect. 3.2 for deducing results about the whole component:

- By way of Theorem 3 and the fact that CtrlEngine(id) is deadlock-free, we deduce the deadlock-freedom of (Engine(id) || CtrlEngine(id)) in the stable failures model.
- By way of Theorem 1 and the fact that the B rewriting of CtrlEngine(id) is consistent, we deduce that (CtrlEngine(id) || Engine(id)) is divergence-free.

```
REFINEMENT CtrlEngine_ref(Id)
VARIABLES
  xpos_csp, speed_csp, cb
INVARIANT
  xpos_csp ∈ Positions_csp
∧ speed_csp ∈ Speeds_csp
∧ cb ∈ 0..2
OPERATIONS
CtrlEngine =
  BEGIN
    cb := 1
  END;
CtrlEngine_perceptions =
  BEGIN
    CHOICE
      BEGIN
      xpos_csp ⟵ getXpos ;
      speed_csp ⟵ getSpeed ;
      cb := 2
      END
    OR
      BEGIN
      speed_csp ⟵ getSpeed ;
      xpos_csp ⟵ getXpos;
      cb := 2
      END
    END
  END;
CtrlEngine_actions =
  BEGIN
    ANY accel_csp WHERE
      accel_csp ∈ Accels_csp
    THEN
       setAccel(accel_csp);
       cb := 1
    END
  END
```

Fig. 5. B rewriting of CtrlEngine(id)

4.2 The Driving System

For the driving system whose CSP behaviour is given Fig. 6, there are three modes to function: SOLO, LEADER or PLATOON. In the SOLO mode, it receives an acceleration from the pilot via the HCI passed on through hciAccel.id ? accel and sends this desired

```
CtrlDrivingSystem(mode,id) =
    ((mode == SOLO) ∨ (mode == LEADER) &
    hciAccel.id ? accel → engineAccel.id ! accel → please_compress(CtrlDrivingSystem(mode,id)) )
    □
   ((mode == PLATOON) &
   getInfluences ? accel → engineAccel.id ! accel → please_compress(CtrlDrivingSystem(mode,id)) )
   □
   ((mode == SOLO) &
   engineInfo.id ? myXpos ? mySpeed →
   hciSpeed.id ! mySpeed → please_compress(CtrlDrivingSystem(mode,id)) )
   □
   ((mode == LEADER) &
   engineInfo.id ? myXpos ? mySpeed →
   hciSpeed.id ! mySpeed → comOut.id ! mySpeed ! myXpos → please_compress(CtrlDrivingSystem(mode,id)) )
   □
   ((mode == PLATOON) &
   engineInfo.id ? myXpos ? mySpeed →
   hciSpeed.id ! mySpeed → comOut.id ! mySpeed ! myXpos → comIn.id ? preSpeed ? preXpos →
   setPerceptions! myXpos ! mySpeed ! preXpos ! preSpeed → please_compress(CtrlDrivingSystem(mode,id)) )
```

Fig. 6. The CtrlDrivingSystem(mode,id) CSP Controller

acceleration to the engine through engineAccel.id ! accel. It can also request Cristal information from the engine via engine Info . id ? myXpos ? mySpeed so as to make the HCI display it (hciSpeed.id ! mySpeed).

The LEADER mode is very similar to the SOLO mode. The only difference concerns an additional sending of the Cristal information to another Cristal via comOut.id ! my Speed ! myXpos.

The PLATOON mode is the mode that actually makes use of a DrivingSystem B machine not given here: acceleration is obtained by a call to the getInfluences method and the result is passed on the engine. The data required for the machine to compute an accurate speed are obtained from the engine (engineInfo.id ? myXpos ? mySpeed) and the leading Cristal comIn.id ? preSpeed ? preXpos. Once the data is obtained, it is passed on to the B machine through the setPerceptions method.

The whole component parametrised by the Cristal identifier and its chosen mode is defined as:

$$(\text{CtrlDrivingSystem(mode,id)} \parallel \text{DrivingSystem(id)})$$

Verification. For the driving system the properties to check are the same as for the engine component:

- The DrivingSystem(id) B machine is consistent.
- For every possible mode, the CtrlDrivingSystem(mode,id) CSP controller is deadlock-free in the stable failures model, and it is divergence-free.
- (CtrlDrivingSystem(mode,id)||DrivingSystem(id)) is deadlock-free.
- (CtrlDrivingSystem(mode,id)||DrivingSystem(id)) is divergence-free.

Note 1. At this point of the models development, verifications become time-consuming for the CSP specifications. The way the processes were modelled (especially for the driving system) made FDR2 take a long time to check deadlock-freedom, for instance. We thus use the FDR2 “compression functions” feature which gives means to speedup

the checking. These functions have no influence on the model itself, but on the way FDR2 explores state space: FDR2 attempts to shrink the state space with specific techniques which may be more or less fruitful depending on the nature of the model [16]. Using compression gives us interesting speedups in verifying the CSP models from there.

4.3 The Assembly Cristal(mode,id)

As illustrated in Fig. 2, a Cristal is defined as the composition of the engine and the driving system:

$$\mathsf{Cristal(mode,id)} = (\mathsf{CtrlEngine(id)} \parallel \mathsf{Engine(id)}) \underset{\{|\mathsf{engineInfo},\ \mathsf{engineAccel}|\}}{\|} (\mathsf{CtrlDrivingSystem(mode,id)} \parallel \mathsf{DrivingSystem(id)})$$

Verification. Divergence-freedom is obtained by applying Theorem 2 to the divergence-freedom of both components (CtrlEngine(id) || Engine(id)) and (CtrlDrivingSystem(mode, id) || DrivingSystem(id)).

Deadlock-freedom of the Cristal stems from deadlock-freedom of (CtrlEngine(id) || CtrlDrivingSystem(mode,id)) (the controllers alone) and applying Theorem 4 to the controllers accompanied by their B machines. Deadlock-freedom as verified by FDR2 is *not* guaranteed for Cristal(mode,id). FDR2 gives some trace examples leading to a deadlock. For instance, a deadlock happens when the engine attempts to send information to the driving system engineInfo.id ! xpos ! speed while the driving system attempts to send an acceleration to the engine engineAccel.id ! accel.

More generally, deadlocks are due to differing expectations from the engine and the driving system: the engine was attempting to send information while the driving system was attempting to send an acceleration, or the engine was expecting an acceleration while the driving system was expecting the Cristal information. This suggested the need for a tighter scheduling of the communications between components.

CtrlDrivingSystem Revisited. To establish deadlock-freedom and fix the problem above, the CSP controller of the driving system has been modified. In fact, the new version of the driving system imposes a scheduling of the process. In the same way as the engine alternates between sending information and receiving a new acceleration, the driving system alternates between receiving information, for dispatching it to the HCI or to the automated driving system, and sending new accelerations, obtained from the HCI or from the automated driving system. The new CtrlDrivingSystem2(mode,id) CSP controller is given Fig. 7.

As previously, the divergence-freedom is obtained through Theorem 2 and divergence-freedom of both CSP||B compounds. Moreover, the deadlock-freedom checking is successful this time: CtrlEngine(id) || CtrlDrivingSystem2(mode,id) is deadlock-free, hence by Theorem 4 Cristal2(mode,id) is deadlock-free. This verification achieves the requirement expressed at the beginning of Sect. 4 where we specified that the communications between components inside a vehicle should not deadlock. Cristal2 is the same as Cristal but with the corrected driving system.

```
CtrlDrivingSystem_perceptions(mode,id) =
  ((mode == SOLO) &
  engineInfo.id ? myXpos ? mySpeed →
  hciSpeed.id ! mySpeed → please_compress(CtrlDrivingSystem_actions(mode,id)) )
  □
  ((mode == LEADER) &
  engineInfo.id ? myXpos ? mySpeed →
  hciSpeed.id ! mySpeed → comOut.id ! mySpeed ! myXpos → please_compress(CtrlDrivingSystem_actions(mode,id)) )
  □
  ((mode == PLATOON) &
  engineInfo.id ? myXpos ? mySpeed →
  hciSpeed.id ! mySpeed → comOut.id ! mySpeed ! myXpos → comIn.id ? preSpeed ? preXpos →
  setPerceptions! myXpos ! mySpeed ! preXpos ! preSpeed → please_compress(CtrlDrivingSystem_actions(mode,id)) )

CtrlDrivingSystem_actions(mode,id) =
   ((mode == SOLO) ∨ (mode == LEADER) &        -- new accel from user
   hciAccel.id ? accel → engineAccel.id ! accel → please_compress(CtrlDrivingSystem_perceptions(mode,id)) )
   □
  ((mode == PLATOON) &        -- new accel from DECISION
  getInfluences ? accel → engineAccel.id ! accel → please_compress(CtrlDrivingSystem_perceptions(mode,id)) )

CtrlDrivingSystem2(mode,id) = CtrlDrivingSystem_perceptions(mode,id)
```

Fig. 7. The CSPCtrlDrivingSystem controller revisited

Safety Property. The safety property we informally expressed at the beginning of Sect. 4 stated that perception and reaction should alternate while the Cristal functions. We can rephrase it here more precisely as the fact that the data – speed and position – are always updated (engineInfo) before applying an instantaneous acceleration to the engine (engineAccel). This ordering of events should constitute a cycle. This property is captured as a CSP process:

$$\mathsf{Property(id)} = \mathsf{engineInfo.id?xpos?speed} \rightarrow \mathsf{engineAccel.id?accel} \rightarrow \mathsf{Property(id)}$$

We need to show that the Cristal meets this specification. For that, we successfully check – using FDR2 – that there is a trace refinement between the CSP part of Cristal2 and Property, i.e. $\mathsf{Property(id)} \sqsubseteq_T \mathsf{CtrlEngine(id)} \parallel \mathsf{CtrlDrivingSystem2(mode,id)}$. Then by Corollary 1 we obtain $\mathsf{Property(id)} \sqsubseteq_T \mathsf{Cristal2\,(mode,id)}$, i.e. the property is satisfied.

5 Specifying a Platoon of Cristals

Once we dispose of a correct model for a single Cristal, we can focus on the specification of a platoon, as shown Fig. 8. We want the various Cristals to avoid going stale when they are in the PLATOON mode. This might happen because one Cristal waits for information from its leading Cristal, for instance. In other words, we do not want the communications in the convoy to deadlock. This is what we will strive for as a safety property for the platoon.

5.1 A Communication Medium

Communications between two successive Cristals are managed at a new layer. Consequently, a new component, called Net(id,id2), is added to each Cristal for managing

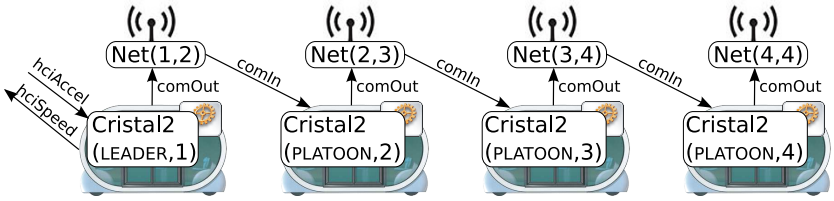

Fig. 8. A Platoon of four Cristals

communication. This communication medium receives the speed and the position from the Cristal identified by id before sending these data to the next Cristal identified by id2. Net(id,id2) is defined by the CSP process given Fig. 9. When the Cristal has no successor in the platoon, Net(id,id2) only consumes the data.

```
Net(id,id2) =
( (id != id2) & comOut.id ? speed ? xpos → comIn.id2 ! speed ! xpos → Net(id,id2) )
□
( (id == id2) & comOut.id ? speed ? xpos → Net(id,id2) )
```

Fig. 9. CSP model Net(id,id2)

Using FDR2, we successfully check that Net(id,id2) is deterministic, deadlock-free in the stable failures model and divergence-free.

5.2 A Platoon of Cristals

A platoon of n Cristals is defined as the parallel composition of n Cristals and n communication mediums.

- The first Cristal of the platoon functions in the LEADER mode, while the others function in the PLATOON mode. The Cristals are independent from each other, consequently their composition is specified using the interleaving operator.

$$\text{Cristals(n)} = \text{Cristal2(LEADER,1)} \;|||\; \left(\underset{\text{id:\{2..n\}}}{|||} \text{Cristal2(PLATOON,id)} \right)$$

- In the platoon, a Net component is associated with each Cristal. Since these components are independent from each other, their composition is specified by interleaving.

$$\text{Nets(n)} = \left(\underset{\text{id:\{1..n-1\}}}{|||} \text{Net(id,id+1)} \right) \;|||\; \text{Net(n,n)}$$

– Finally, the platoon is defined by the parallel composition of all the Cristals and all the Nets, synchronised on {|comIn, comOut|}.

$$\mathsf{Platoon(n)} = \mathsf{Cristals(n)} \underset{\{|\mathsf{comIn},\ \mathsf{comOut}|\}}{||} \mathsf{Nets(n)}$$

Verification. As each Cristal and each Net have been proved divergence-free, the platoon is divergence-free by applying Theorem 2. To achieve consistency checking, the parallel composition of the CSP parts of each Cristal and communication medium is shown deadlock-free, thanks to FDR2. Consequently, by Theorem 4 the platoon is deadlock-free too. This verification validates the safety property expressed at the beginning of Sect. 5 saying that the communications (expressed through the Nets components) should not deadlock.

6 Conclusion

The development of a new type of urban vehicle and the needs for its certification necessitate their formal specification and validation. We propose in this paper a formal CSP||B specification development of an autonomous vehicle components, and an architecture for assembling vehicles in a convoy to follow the path of the leader vehicle in a row. Applying known results on the composition and the verification in the CSP||B framework and using existing tools, the FDR2 model-checking and the B4Free proof tools, allow us to ensure the consistency of the whole multi-agent system, in a compositional manner.

Having formal CSP||B specifications help – by establishing refinement relations – to prevent incompatibility among various implementations. Moreover, writing formal specifications help in designing a way to manage the multi-level assembly.

This work points out the main drawback of the CSP||B approach: at the interface between the two models, CLIs and augmented B machines corresponding to CSP controllers are not automatically generated. But this task requires a high expertise level. In our opinion, the user should be able to conduct all the verification steps automatically. Automation of these verification steps could be a direction for future work.

On the case study side, to go further, we are currently studying new properties such as the non-collision, the non-unhooking and the non-oscillation: which ones are expressible with CSP||B, which ones are tractable and verifiable? This particular perspective is related to a similar work by the authors of CSP||B who dealt with another kind of multi-agent system in [14]. So far our use of CSP||B for the platooning model reaches similar conclusions. This nonetheless begs the question of which impact the expression of more complex emerging properties does have on the model.

Further model development requires checking other refinement relations. It also includes evolutions in order to study what happens when a Cristal joins or leaves the platoon, and which communication protocols must be obeyed to do so in a safe manner. We also plan to take into account the lateral control and/or perturbations such as pedestrians or other vehicles.

Acknowledgement. We would like to thank Olivier Simonin, Alexis Scheuer and François Charpillet from the LORIA/MAIA team for common efforts and fruitful discussions in the context of the TACOS and the CRISTAL projects.

References

1. Schneider, S.A., Treharne, H.E.: CSP theorems for communicating B machines. Formal Aspects of Computing, Special issue of IFM'04 (2005)
2. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
3. Abrial, J.R.: The B Book. Cambridge University Press (1996)
4. Simonin, O., Lanoix, A., Colin, S., Scheuer, A., Charpillet, F.: Generic Expression in B of the Influence/Reaction Model: Specifying and Verifying Situated Multi-Agent Systems. INRIA Research Report 6304, INRIA (2007)
5. Bontron, P., Potet, M.L.: Automatic construction of validated B components from structured developments. In: Proc. First Int. Conf. ZB'2000, York, Great Britain. Volume 1878 of LNCS., Springer Verlag (2000) 127–147
6. Abrial, J.R.: Discrete system models. Version 1.1 (2002)
7. Attiogbé, J.: Communicating B abstract systems. Research Report RR-IRIN 02.08 (2002) updated july 2003.
8. Bellegarde, F., Julliand, J., Kouchnarenko, O.: Synchronized parallel composition of event systems in B. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: Formal specification and development in Z and B (ZB'2002). Volume 2272 of LNCS., Springer-Verlag (2002) 436–457
9. Daviet, P., Parent, M.: Longitudinal and lateral servoing of vehicles in a platoon. In: Proceeding of the IEEE Intelligent Vehicles Symposium. (1996) 41–46
10. Ferber, J., Muller, J.P.: Influences and reaction : a model of situated multiagent systems. In: 2nd Int. Conf. on Multi-agent Systems. (1996) 72–79
11. Treharne, H., Schneider, S.: Using a process algebra to control B OPERATIONS. In: 1st International Conference on Integrated Formal Methods (IFM'99), York, Springer Verlag (1999) 437–457
12. Schneider, S., Treharne, H.: Communicating B machines. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: Formal specification and development in Z and B (ZB 2002). Volume 2272 of LNCS., Springer Verlag (2002) 416–435
13. Evans, N., Treharne, H.E.: Investigating a file transfer protocol using CSP and B. Software and Systems Modelling Journal **4** (2005) 258–276
14. Schneider, S., Cavalcanti, A., Treharne, H., Woodcock, J.: A layered behavioural model of platelets. In: 11th IEEE International Conference on Engieerging of Complex Computer Systems, ICECCS. (2006)
15. Roscoe, A.W.: The theory and Practice of Concurrency. Prentice Hall (1997)
16. Formal Systems (Europe) Ltd.: Failures-Divergence Refinement – FDR2 user manual. Formal systems (europe) ltd. edn. (1997) Available at `http://www.formal.demon.co.uk/fdr2manual/index.html`.

A Specifications of a Single Cristal

A.1 Global definitions

```
MODEL
   Constants
CONSTANTS
  MAX_SPEED,
  MIN_ACCEL,
  MAX_ACCEL,
  ALERT_DISTANCE,
  IDEAL_DISTANCE
PROPERTIES
  MAX_SPEED ∈ NAT1 ∧
  MIN_ACCEL ∈ INT ∧
  MIN_ACCEL < 0 ∧
  MAX_ACCEL ∈ NAT1 ∧
  ALERT_DISTANCE ∈ NAT ∧
  IDEAL_DISTANCE ∈ NAT ∧
  ALERT_DISTANCE < IDEAL_DISTANCE
ASSERTIONS
  ∀(i , j ).(  (i  ∈ ℤ ∧ j ∈ ℤ ∧ i ≤ j)  ⇒
    (∀k.((k ∈ ℤ) ⇒ ( min({j ,max({i,k })})  ∈ i .. j  ))  )   )
END
```

```
datatype Modes = PLATOON | LEADER | SOLO
datatype Params = TRANSMIT | LAST
MAX_ID = 10
nametype Ids = {1..MAX_ID}
MAX_SPEED = 1
MIN_ACCEL = −1
MAX_ACCEL = 1
MAX_POS = 1
UNHOOKING_DIST = 1
nametype Speeds = {0..MAX_SPEED}
nametype Accels = {MIN_ACCEL..MAX_ACCEL}
nametype Positions = {0..MAX_POS}
−− Common channels between CtrlEngine∧CtrlDrivingSystem
channel engineInfo: Ids. Positions . Speeds
channel engineAccel: Ids. Accels

−− B machine channels between Engine∧CtrlEngine
channel getSpeed: Speeds
channel getXpos: Positions
channel setAccel: Accels

−− B machine channels between DrivingSystem∧CtrlDrivingSystem
channel setPerceptions : Positions . Speeds . Positions . Speeds
channel getInfluences : Accels

−− Channels between  an HCI∧CtrlDrivingSystem
channel hciAccel :  Ids. Accels
channel hciSpeed :  Ids. Speeds

−− Channels between other cristals∧CtrlDrivingSystem
channel comIn : Ids. Speeds . Positions
channel comOut : Ids. Speeds . Positions

InternalEngine = {| getSpeed, getXpos,setAccel|}
InternalDrivingSystem = {|setPerceptions, getInfluences|}
EngineDrivingSystem = {| engineInfo, engineAccel |}
Environment = {| hciAccel,  hciSpeed, comIn, comOut |}
Cristal_NotEngine = union(InternalEngine, union(InternalDrivingSystem, Environment))
All_Channels = union(EngineDrivingSystem, Cristal_NotEngine)
```

A.2 CtrlEngine(id)||Engine(id)

```
MODEL Engine(Id)
CONSTRAINTS Id ∈ NAT1
SEES Constants
VARIABLES
  speed, xpos
INVARIANT
  speed ∈ 0..MAX_SPEED
∧ xpos ∈ ℕ
INITIALISATION
  speed := 0 ‖ xpos :∈ ℕ
OPERATIONS
  speed0 ⟵ getSpeed = /*...*/
    BEGIN
      speed0 := speed
    END ;
  xpos0 ⟵ getXpos = /*...*/
    BEGIN
      xpos0 := xpos
    END ;
  setAccel(accel) =
    PRE
      accel ∈ MIN_ACCEL..MAX_ACCEL
    THEN
      ANY new_speed
      WHERE new_speed = speed + accel
      THEN
        IF (new_speed > MAX_SPEED)
        THEN
          xpos := xpos + MAX_SPEED
          ‖ speed := MAX_SPEED
        ELSE
          IF (new_speed < 0)
          THEN
            xpos := xpos − (speed × speed) / (2 × accel)
            ‖ speed := 0
          ELSE
            xpos := xpos + speed + accel / 2
            ‖ speed := new_speed
          END
        END
      END
    END
END
```

```
CtrlEngine_perceptions(id) =
  getXpos ? xpos → getSpeed ? speed → engineInfo.id ! xpos ! speed → CtrlEngine_actions(id)
  □
  getSpeed ? speed → getXpos ? xpos → engineInfo.id ! xpos ! speed → CtrlEngine_actions(id)

CtrlEngine_actions(id) =
  engineAccel.id ? accel → setAccel ! accel → CtrlEngine_perceptions(id)

CtrlEngine(id) = CtrlEngine_perceptions(id)
```

```
assert CtrlEngine(1) :[ deadlock free [F] ]
assert CtrlEngine(1) :[ divergence free ]
assert CtrlEngine(1) \ InternalEngine :[ divergence free ]
```

```
MACHINE CtrlEngine_abs(Id)
CONSTRAINTS
  Id ∈ NAT1
VARIABLES
  cb
```

```
INVARIANT
  cb ∈ 0..2
INITIALISATION
  cb := 0
OPERATIONS
CtrlEngine =
  PRE cb = 0
  THEN
    cb :∈ 0..2
  END;
CtrlEngine_perceptions =
  PRE cb = 1
  THEN
    cb :∈ 0..2
  END;
CtrlEngine_actions =
  PRE cb = 2
  THEN
    cb :∈ 0..2
  END
END
```

```
REFINEMENT CtrlEngine_ref(Id)
REFINES CtrlEngine_abs
INCLUDES Engine(Id)
SEES
  Constants_csp, Constants
VARIABLES
  xpos_csp, speed_csp, cb
INVARIANT
  xpos_csp ∈ Positions_csp
∧ speed_csp ∈ Speeds_csp
∧ cb ∈ 0..2
INITIALISATION
   xpos_csp :∈ Positions_csp
|| speed_csp :∈ Speeds_csp
|| cb := 0
OPERATIONS
CtrlEngine =
  BEGIN
    cb := 1
  END;
CtrlEngine_perceptions =
  BEGIN
    CHOICE
      BEGIN
      xpos_csp ⟵ getXpos ;
      speed_csp ⟵ getSpeed ;
      cb := 2
      END
    OR
      BEGIN
      speed_csp ⟵ getSpeed ;
      xpos_csp ⟵ getXpos;
      cb := 2
      END
    END
  END;
CtrlEngine_actions =
  BEGIN
    ANY accel_csp WHERE
      accel_csp ∈ Accels_csp
    THEN
       setAccel(accel_csp);
       cb := 1
    END
  END
```

```
END
```

A.3 CtrlDrivingSystem(mode,id)||DrivingSystem(id)

```
MODEL DrivingSystem(Id)
CONSTRAINTS Id ∈ NAT1
SEES Constants
VARIABLES
  myXpos, mySpeed,
  preXpos, preSpeed
INVARIANT
  myXpos ∈ ℕ
∧ mySpeed ∈ 0..MAX_SPEED
∧ preXpos ∈ ℕ
∧ preSpeed ∈ 0..MAX_SPEED
INITIALISATION
  myXpos :∈ ℕ
|| mySpeed := 0
|| preXpos :∈ ℕ
|| preSpeed := 0
OPERATIONS
  setPerceptions(myXpos0, mySpeed0, preXpos0, preSpeed0) =
    PRE
      myXpos0 ∈ ℕ
      ∧ mySpeed0 ∈ 0..MAX_SPEED
      ∧ preXpos0 ∈ ℕ
      ∧ preSpeed0 ∈ 0..MAX_SPEED
    THEN
      myXpos := myXpos0
      || mySpeed := mySpeed0
      || preXpos := preXpos0
      || preSpeed := preSpeed0
    END ;
  accel ⟵ getInfluences =
    IF (preXpos − myXpos < ALERT_DISTANCE)
    THEN
      accel := MIN_ACCEL
    ELSE
      ANY new_accel
      WHERE
        new_accel = 2 × (preXpos − myXpos) − IDEAL_DISTANCE + preSpeed − mySpeed
      THEN
        accel := min( {MAX_ACCEL, max( {MIN_ACCEL, new_accel} ) } )
      END
    END
END
```

```
CtrlDrivingSystem(mode,id) =
    ((mode == SOLO) ∨ (mode == LEADER) &
    hciAccel.id ? accel → engineAccel.id ! accel → please_compress(CtrlDrivingSystem(mode,id)) )
    □
  ((mode == PLATOON) &
  getInfluences ? accel → engineAccel.id ! accel → please_compress(CtrlDrivingSystem(mode,id)) )
  □
  ((mode == SOLO) &
  engineInfo.id ? myXpos ? mySpeed →
  hciSpeed.id ! mySpeed → please_compress(CtrlDrivingSystem(mode,id)) )
  □
  ((mode == LEADER) &
  engineInfo.id ? myXpos ? mySpeed →
  hciSpeed.id ! mySpeed → comOut.id ! mySpeed ! myXpos → please_compress(CtrlDrivingSystem(mode,id)) )
  □
  ((mode == PLATOON) &
  engineInfo.id ? myXpos ? mySpeed →
```

```
  hciSpeed.id ! mySpeed → comOut.id ! mySpeed ! myXpos → comIn.id ? preSpeed ? preXpos →
  setPerceptions! myXpos ! mySpeed ! preXpos ! preSpeed → please_compress(CtrlDrivingSystem(mode,id)) )
```

```
assert CtrlDrivingSystem(SOLO,1) :[ deadlock free [F] ]
assert CtrlDrivingSystem(LEADER,1) :[ deadlock free [F] ]
assert CtrlDrivingSystem(PLATOON,1) :[ deadlock free [F] ]
assert CtrlDrivingSystem(SOLO,1) :[ divergence free ]
assert CtrlDrivingSystem(LEADER,1) :[ divergence free ]
assert CtrlDrivingSystem(PLATOON,1) :[ divergence free ]
assert CtrlDrivingSystem(SOLO,1) \ InternalDrivingSystem :[ divergence free ]
assert CtrlDrivingSystem(LEADER,1) \ InternalDrivingSystem :[ divergence free ]
assert CtrlDrivingSystem(PLATOON,1) \ InternalDrivingSystem :[ divergence free ]
```

```
MACHINE CtrlDrivingSystem_abs(Mode,Id)
CONSTRAINTS
 Id ∈ NAT1 ∧ Mode ∈ 1..3
VARIABLES
  cb
INVARIANT
  cb ∈ 0..0
INITIALISATION
  cb := 0
OPERATIONS
CtrlDrivingSystem =
  PRE cb = 0
  THEN cb :∈ 0..0
  END
END
```

```
REFINEMENT CtrlDrivingSystem_ref(Mode,Id)
REFINES CtrlDrivingSystem_abs
SEES Constants_csp, Constants
INCLUDES DrivingSystem(Id)
VARIABLES
  accel, cb
INVARIANT
  accel ∈ Accels_csp
∧ cb ∈ 0..0
INITIALISATION
  accel := 0
|| cb := 0
OPERATIONS
CtrlDrivingSystem =
  BEGIN
    CHOICE
      IF Modes_csp_of_nat(Mode) = SOLO ∨ Modes_csp_of_nat(Mode) = LEADER
      THEN
        ANY ihmAccel_accel
        WHERE ihmAccel_accel ∈ Accels_csp
        THEN cb := 0
        END
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    OR
      IF Modes_csp_of_nat(Mode) = PLATOON
      THEN
        accel ⟵ getInfluences;
        cb := 0
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    OR
      IF Modes_csp_of_nat(Mode) = SOLO
      THEN
        ANY motorInfo_myXpos, motorInfo_mySpeed
        WHERE motorInfo_myXpos ∈ Positions_csp
            ∧ motorInfo_mySpeed ∈ Speeds_csp
```

```
        THEN cb := 0
        END
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    OR
      IF Modes_csp_of_nat(Mode) = LEADER
      THEN
        ANY motorInfo_myXpos, motorInfo_mySpeed
        WHERE motorInfo_myXpos ∈ Positions_csp
            ∧ motorInfo_mySpeed ∈ Speeds_csp
        THEN cb := 0
        END
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    OR
      IF Modes_csp_of_nat(Mode) = PLATOON
      THEN
        ANY motorInfo_myXpos, motorInfo_mySpeed
        WHERE motorInfo_myXpos ∈ Positions_csp
            ∧ motorInfo_mySpeed ∈ Speeds_csp
        THEN
          ANY comIn_preSpeed, comIn_preXpos
          WHERE comIn_preSpeed ∈ Speeds_csp
              ∧ comIn_preXpos ∈ Positions_csp
          THEN
            setPerceptions(motorInfo_myXpos, motorInfo_mySpeed,
                           comIn_preXpos, comIn_preSpeed);
            cb := 0
          END
        END
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    END
  END
END
```

A.4 Cristal(mode,id)

```
Cristal_verif (mode,id) = (CtrlDrivingSystem(mode,id) [| EngineDrivingSystem |] CtrlEngine(id))
```

```
assert Cristal_verif (SOLO,1) :[ deadlock free [F] ]
assert Cristal_verif (LEADER,1) :[ deadlock free [F] ]
assert Cristal_verif (PLATOON,1) :[ deadlock free [F] ]
```

A.5 CtrlDrivingSystem2(mode,id)||DrivingSystem(id)

```
CtrlDrivingSystem_perceptions(mode,id) =
  ((mode == SOLO) &
  engineInfo.id ? myXpos ? mySpeed →
  hciSpeed.id ! mySpeed → please_compress(CtrlDrivingSystem_actions(mode,id)) )
  □
  ((mode == LEADER) &
  engineInfo.id ? myXpos ? mySpeed →
  hciSpeed.id ! mySpeed → comOut.id ! mySpeed ! myXpos → please_compress(CtrlDrivingSystem_actions(mode,id)) )
  □
  ((mode == PLATOON) &
  engineInfo.id ? myXpos ? mySpeed →
  hciSpeed.id ! mySpeed → comOut.id ! mySpeed ! myXpos → comIn.id ? preSpeed ? preXpos →
  setPerceptions! myXpos ! mySpeed ! preXpos ! preSpeed → please_compress(CtrlDrivingSystem_actions(mode,id)) )
```

```
CtrlDrivingSystem_actions(mode,id) =
    ((mode == SOLO) ∨ (mode == LEADER) &          −− new accel from user
    hciAccel.id ? accel → engineAccel.id ! accel → please_compress(CtrlDrivingSystem_perceptions(mode,id)) )
    □
   ((mode == PLATOON) &          −− new accel from DECISION
   getInfluences ? accel → engineAccel.id ! accel → please_compress(CtrlDrivingSystem_perceptions(mode,id)) )

CtrlDrivingSystem2(mode,id) = CtrlDrivingSystem_perceptions(mode,id)
```

```
assert CtrlDrivingSystem2(SOLO,1) :[ deadlock free [F] ]
assert CtrlDrivingSystem2(LEADER,1) :[ deadlock free [F] ]
assert CtrlDrivingSystem2(PLATOON,1) :[ deadlock free [F] ]
assert CtrlDrivingSystem2(SOLO,1) :[ divergence free ]
assert CtrlDrivingSystem2(LEADER,1) :[ divergence free ]
assert CtrlDrivingSystem2(PLATOON,1) :[ divergence free ]
assert CtrlDrivingSystem2(SOLO,1) \ InternalDrivingSystem :[ divergence free ]
assert CtrlDrivingSystem2(LEADER,1) \ InternalDrivingSystem :[ divergence free ]
assert CtrlDrivingSystem2(PLATOON,1) \ InternalDrivingSystem :[ divergence free ]
```

```
MACHINE CtrlDrivingSystem2_abs(Mode,Id)
CONSTRAINTS
    Id ∈ NAT1
∧ Mode ∈ 1..3
VARIABLES
  cb
INVARIANT
  cb ∈ 0..2
INITIALISATION
  cb := 0
OPERATIONS
CtrlDrivingSystem2 =
  PRE cb = 0
  THEN cb :∈ 0..2
  END;
CtrlDrivingSystem_perceptions =
  PRE cb = 1
  THEN cb :∈ 0..2
  END;
CtrlDrivingSystem_actions =
  PRE cb = 2
  THEN cb :∈ 0..2
  END
END
```

```
REFINEMENT CtrlDrivingSystem2_ref(Mode,Id)
REFINES CtrlDrivingSystem2_abs
SEES Constants_csp, Constants
INCLUDES DrivingSystem(Id)
VARIABLES
  accel, cb
INVARIANT
  accel ∈ Accels_csp
∧ cb ∈ 0..2
INITIALISATION
  accel := 0
|| cb := 0
OPERATIONS
CtrlDrivingSystem2 =
  BEGIN
    cb := 1
  END;
CtrlDrivingSystem_perceptions =
  BEGIN
    CHOICE
      IF Modes_csp_of_nat(Mode) = SOLO
      THEN
```

```
      ANY motorInfo_myXpos, motorInfo_mySpeed
      WHERE motorInfo_myXpos ∈ Positions_csp
          ∧ motorInfo_mySpeed ∈ Speeds_csp
      THEN cb := 2
      END
    ELSE SELECT TRUE = FALSE THEN skip END
    END
  OR
    IF Modes_csp_of_nat(Mode) = LEADER
    THEN
      ANY motorInfo_myXpos, motorInfo_mySpeed
      WHERE motorInfo_myXpos ∈ Positions_csp
          ∧ motorInfo_mySpeed ∈ Speeds_csp
      THEN cb := 2
      END
    ELSE SELECT TRUE = FALSE THEN skip END
    END
  OR
    IF Modes_csp_of_nat(Mode) = PLATOON
    THEN
      ANY motorInfo_myXpos, motorInfo_mySpeed
      WHERE motorInfo_myXpos ∈ Positions_csp
          ∧ motorInfo_mySpeed ∈ Speeds_csp
      THEN
        ANY comIn_preSpeed, comIn_preXpos
        WHERE comIn_preSpeed ∈ Speeds_csp
            ∧ comIn_preXpos ∈ Positions_csp
        THEN
          setPerceptions(motorInfo_myXpos, motorInfo_mySpeed,
                         comIn_preXpos, comIn_preSpeed);
          cb := 2
        END
      END
    ELSE SELECT TRUE = FALSE THEN skip END
    END
  END
END;
CtrlDrivingSystem_actions =
  BEGIN
    CHOICE
      IF Modes_csp_of_nat(Mode) = SOLO ∨ Modes_csp_of_nat(Mode) = LEADER
      THEN
        ANY ihmAccel_accel
        WHERE ihmAccel_accel ∈ Accels_csp
        THEN cb := 1
        END
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    OR
      IF Modes_csp_of_nat(Mode) = PLATOON
      THEN
        accel ⟵ getInfluences;
        cb := 1
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    END
  END
END
```

A.6 Cristal2(mode,id)

```
Cristal2_verif (mode,id) = (CtrlDrivingSystem2(mode,id) [| EngineDrivingSystem |] CtrlEngine(id))
```

```
assert Cristal2_verif (SOLO,1) :[ deadlock free [F] ]
assert Cristal2_verif (LEADER,1) :[ deadlock free [F] ]
assert Cristal2_verif (PLATOON,1) :[ deadlock free [F] ]
```

```
Property(id) = engineInfo.id?xpos?speed → engineAccel.id?accel → Property(id)
```

```
Cristal2_EDS(mode,id) = Cristal2_verif(mode,id) \ Cristal_NotEngine
```

```
assert Property(1) ⊑_T Cristal2_EDS(SOLO,1)
assert Property(1) ⊑_T Cristal2_EDS(LEADER,1)
assert Property(1) ⊑_T Cristal2_EDS(PLATOON,1)
```

B Specifications of a Platoon of Cristals

B.1 Cristals

```
OnlyCom = {| engineInfo, engineAccel, setPerceptions, getInfluences, getSpeed, getXpos,setAccel, hciAccel, hciSpeed |}

Cristal_p(mode, id) = please_compress( (CtrlEngine(id) [| {|engineInfo, engineAccel|} |] CtrlDrivingSystem2(mode,id) ) \ OnlyCom )

Cristals(max) = Cristal_p(LEADER, 1) ||| ( ||| id :{2.. max} @ Cristal_p(PLATOON, id) )
```

B.2 Net

```
Net(id,id2) =
( (id != id2) & comOut.id ? speed ? xpos → comIn.id2 ! speed ! xpos → Net(id,id2) )
□
( (id == id2) & comOut.id ? speed ? xpos → Net(id,id2) )
```

```
assert Net(2,3) :[ deadlock free [F] ]
assert Net(2,3) :[ divergence free ]
assert Net(5,5) :[ deadlock free [F] ]
assert Net(5,5) :[ divergence free ]
```

B.3 Nets

```
Nets(max) = ( ||| id :{1.. max−1} @ Net(id,id+1) ) ||| Net(max,max)
```

B.4 Platoon of Cristals

```
Platoon_verif(max) = Cristals(max) [|{| comIn, comOut|}|] Nets(max)
```

```
assert Platoon_verif(5) :[ deadlock free [F] ]
assert Platoon_verif(5) :[ divergence free ]
```